Приложения на сортиращите алгоритми

Тодор Дуков

Обща информация за някои алгоритми за сортиране

Понякога се оказва много удобно да сортираме входните данни, защото това ни носи полезна информация. За това е хубаво да се знаят различните алгоритми за сортиране и в какво те са добри. Нека ги сравним по тяхната сложност, като:

- $T_{avg}(n)$ е сложността по време в средния случай
- $M_{avg}(n)$ е сложността по памет в средния случай
- $T_{worst}(n)$ е сложността по време в най-лошия случай
- $M_{worst}(n)$ е сложността по памет в най-лошия случай

име	$T_{avg}(n)$	$M_{avg}(n)$	$T_{worst}(n)$	$M_{worst}(n)$
пирамидално сортиране	$\Theta(n\log(n))$	$\Theta(1)$	$\Theta(n\log(n))$	$\Theta(1)$
сортиране чрез сливане	$\Theta(n\log(n))$	$\Theta(n)$	$\Theta(n\log(n))$	$\Theta(n)$
бързо сортиране	$\Theta(n\log(n))$	$\Theta(\log(n))$	$\Theta(n^2)$	$\Theta(n)$

Въпреки че алгоритъмът за бързо сортиране е по-бавен в най-лошият случай от този пирамидално сортиране или този за сортиране чрез сливане, практически се оказва, че се справя по-добре. Разбира се другите алгоритми също си имат техните предимства. Пирамидалното сортиране заема малко памет, което е много полезно във вградени системи. Сортирането чрез сливането се оказва по-бърз, когато данните са много големи и се съхраняват във външни устройства (да кажем на твърд диск). Овен тези алгоритми има и други хубави алгоритми като сортиране чрез броене, но за съжаление ние няма да ги разглеждаме тук.

Два алгоритъма

Ще започнем с два често срещани алгоритъма, които се възползват от това, че данните идват сортирани:

- двоично търсене
- алгоритъма за задачата 2-sum

Нека започнем с алгоритъма за двоично търсене:

```
int binary_search(int *arr, int n, int val)
{
    int left = 0, right = n - 1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (arr[mid] == val)
            return mid;
        else if (arr[mid] < val)
            left = mid + 1;
        else
            right = mid - 1;
    }

    return -1;
}
```

При подаден сортиран масив от числа $\texttt{arr}$ с размер $\texttt{n}$ и стойност $\texttt{val}$, функцията $\texttt{binary\_search}(\texttt{arr}, \texttt{n}, \texttt{val})$ ще върне индекс на $\texttt{arr}$, в който се намира $\texttt{val}$, ако има такъв, иначе ще върне -1:

Инварианта. *При всяко достигане на ред 5 имаме, че стойността* $\texttt{val}$ *не се намира измежду двата масива* $\texttt{arr}[0 \ldots \texttt{left} - 1]$ *и* $\texttt{arr}[\texttt{right} + 1 \ldots \texttt{n} - 1]$.

База. При първото достигане имаме, че $\texttt{left} = 0$ и $\texttt{right} = \texttt{n} - 1$. Наистина стойността $\texttt{val}$ не се намира измежду двата масива $\texttt{arr}[0 \ldots 0 - 1]$ и $\texttt{arr}[\texttt{n} - 1 + 1 \ldots \texttt{n} - 1]$.
Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на ред 5. Тогава $\texttt{val}$ не се намира измежду $\texttt{arr}[0 \ldots \texttt{left} - 1]$ и $\texttt{arr}[\texttt{right} + 1 \ldots \texttt{n} - 1]$, и понеже достигането е непоследно, $\texttt{left} \leq \texttt{right}$. Тогава $\texttt{mid} = \left\lfloor \frac{\texttt{left}+\texttt{right}}{2} \right\rfloor$, откъдето $\texttt{left} \leq \texttt{mid} \leq \texttt{right}$. Трябва да разгледаме следните три случая:

1 сл. $\texttt{arr}[\texttt{mid}] = \texttt{val}$ – това няма как да е изпълнено понеже достигането е нефинално

2 сл. $\texttt{arr}[\texttt{mid}] < \texttt{val}$ – понеже $\texttt{arr}$ е сортиран, няма как $\texttt{val}$ да се намира измежду $\texttt{arr}[0 \ldots \texttt{mid}]$, откъдето $\texttt{val}$ не се намира измежду $\texttt{arr}[0 \ldots \underbrace{\texttt{mid} + 1}_{\texttt{left}_{\texttt{new}}} - 1]$ и $\texttt{arr}[\texttt{right} + 1 \ldots \texttt{n} - 1]$.

3 сл. $\texttt{arr}[\texttt{mid}] > \texttt{val}$ – напълно дуален на 2 сл.

Терминация. От цикъла можем да излезем по два начина:

- не е изпълнено условието на ред 5 т.е. $\texttt{left} > \texttt{right}$ – тогава $\texttt{left} - 1 \geq \texttt{right}$ и понеже $\texttt{val}$ не се намира измежду $\texttt{arr}[0 \ldots \texttt{left} - 1]$ и $\texttt{arr}[\texttt{right} + 1 \ldots \texttt{n} - 1]$, $\texttt{val}$ не се намира във $\texttt{arr}[0 \ldots \texttt{n} - 1]$. Накрая алгоритъмът ще върне -1, което наистина е желания резултат.

- изпълнено е условието на ред 9 т.е. $\texttt{arr}[\texttt{mid}] = \texttt{val}$ – тогава алгоритъмът връща $\texttt{mid}$ т.е. точно това, което искаме.

Нека сега разгледаме алгоритъма за задачата 2-sum:

```
1  bool two_sum(int *arr, int n, int target)
2  {
3      int left = 0, right = n - 1;
4  
5      while (left < right)
6      {
7          if (arr[left] + arr[right] == target)
8              return true;
9          else if (arr[left] + arr[right] < target)
10             ++left;
11         else
12             --right;
13     }
14 
15     return false;
16 }
```

Ще покажем, че при подаден сортиран масив $\texttt{arr}$ с размер $\texttt{n}$ и число $\texttt{target}$, функцията $\texttt{two\_sum}(\texttt{arr}, \texttt{n}, \texttt{target})$ разпознава дали има $0 \leq \texttt{i} < \texttt{j} \leq \texttt{n} - 1$, за които:

$$\texttt{arr}[\texttt{i}] + \texttt{arr}[\texttt{j}] = \texttt{target} \quad // \text{ всяка такава двойка } (\texttt{i}, \texttt{j}) \text{ ще наричаме } \textit{диада}$$

Инварианта. *При всяко достигане на ред 5 имаме, че всички диади се намират в* $\texttt{arr}[\texttt{left} \ldots \texttt{right}]$.

База. При първото достигане имаме, че $\texttt{left} = 0$ и $\texttt{right} = \texttt{n} - 1$. Тогава твърдението е тривиално изпълнено.
Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на ред 5. Тогава всички диади се намират в $\texttt{arr}[\texttt{left} \ldots \texttt{right}]$ и понеже достигането е непоследно, $\texttt{left} < \texttt{right}$. Разглеждаме три случая:

1 сл. $\texttt{arr}[\texttt{left}] + \texttt{arr}[\texttt{right}] = \texttt{target}$ – това няма как да е изпълнено понеже достигането е нефинално

2 сл. $\texttt{arr}[\texttt{left}] + \texttt{arr}[\texttt{right}] < \texttt{target}$ – тогава понеже $\texttt{arr}$ е сортиран, за всяко $\texttt{left} + 1 \leq \texttt{i} \leq \texttt{right}$ имаме, че $\texttt{arr}[\texttt{left}] + \texttt{arr}[\texttt{i}] \leq \texttt{arr}[\texttt{left}] + \texttt{arr}[\texttt{right}] < \texttt{target}$. Това означава, че $\texttt{left}$ не участва в никоя диада във $\texttt{arr}[\texttt{left} \ldots \texttt{right}]$. Така получаваме, че всички диади се намират в $\texttt{arr}[\underbrace{\texttt{left} + 1}_{\texttt{left}_{\texttt{new}}} \ldots \texttt{right}]$

3 сл. $\texttt{arr}[\texttt{left}] + \texttt{arr}[\texttt{right}] > \texttt{target}$ – напълно дуален на 2 сл.

Терминация. От цикъла можем да излезем по два начина:

- не е изпълнено условието на ред 5 т.е. $\texttt{left} \geq \texttt{right}$ – тогава няма диади в `arr`, и алгоритъмът коректно ще върне `false`.
- изпълнено е условието на ред 9 т.е. $\texttt{arr}[\texttt{left}] + \texttt{arr}[\texttt{right}] = \texttt{target}$ – тогава алгоритъмът връща `true` т.е. точно това, което искаме.

Двата алгоритъма са доста подобни, използват една често срещана техника – two-pointer. Това е много силен метод за решаване на задачи за търсене в някакво множество от елементи. Започва се с цялото множество и то постепенно се смалява. Разбира се, тук разгледаните алгоритми имат разлика в сложността, поради разликата в стесняването:

- първият алгоритъм има сложност $O(\log(n))$ понеже винаги разликата между `left` и `right` намалява двойно
- вторият алгоритъм има сложност $O(n)$ понеже винаги разликата между `left` и `right` намалява с единица

Задачи

Задача 1. Да се напише колкото се може по-бърз алгоритъм $\texttt{find\_peak}(\texttt{arr}, \texttt{n})$, който приема масив от различни числа `arr` с размер $\texttt{n} \geq 3$, за който има $0 < \texttt{i} < \texttt{n} - 1$ такова, че $\texttt{arr}[0 \dots \texttt{i}]$ е сортиран възходящо и $\texttt{arr}[\texttt{i} \dots \texttt{n} - 1]$ е сортиран низходящо, и връща това `i`. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 2. Да се напише колкото се може по-бърз алгоритъм $\texttt{search\_range}(\texttt{arr}, \texttt{n}, \texttt{val})$, който при подаден сортиран масив `arr` с размер `n` и число `val`, което се намира в масива, връща най-малкият и най-големият индекс, на който `val` се намира в `arr`. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 3. Да се напише колкото се може по-бърз алгоритъм $\texttt{k\_sum}(\texttt{arr}, \texttt{n}, \texttt{k}, \texttt{target})$, който при подаден сортиран масив `arr` с размер `n` и числа $\texttt{k} \geq 2$ и `target`, връща дали има $0 \leq \texttt{i}_1 < \texttt{i}_2 < \dots < \texttt{i}_\texttt{k} \leq \texttt{n} - 1$, за които:

$$\texttt{arr}[\texttt{i}_1] + \texttt{arr}[\texttt{i}_2] + \dots + \texttt{arr}[\texttt{i}_\texttt{k}] = \texttt{target}$$

След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4. Да се напише колкото се може по-бърз алгоритъм $\texttt{k\_closest}(\texttt{arr}, \texttt{n}, \texttt{k}, \texttt{target})$, който принтира най-близките `k` числа в `arr` до `target`. След това да се докаже неговата коректност, и да се изследва сложността му по време. Може ли да се напише по-бърз алгоритъм при предположение че `arr` е сортиран?

Задача 5. Да се напише колкото се може по-бърз алгоритъм $\texttt{sort}(\texttt{arr}, \texttt{n})$, който приема масив `arr` с размер `n`, съставен от числата $0, 1$ и 2, и го сортира. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6. Да се напише колкото се може по-бърз алгоритъм $\texttt{merge\_intervals}(\texttt{intervals}, \texttt{n})$, който приема масив `intervals` с размер `n`, съставен от двойки числа, които представят някакъв затворен интервал, и връща нов масив, в който са сляти всички интервали с непразно сечение. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 7. Да се напише колкото се може по-бърз алгоритъм $\texttt{sorted\_squares}(\texttt{arr}, \texttt{n})$, който приема масив `arr` с размер `n`, и връща нов масив от квадратите на `arr`, който е сортиран. След това да се докаже неговата коректност, и да се изследва сложността му по време.